SYSTEM Command

FORMAT:

```
SYSTEM
```

PURPOSE:

Leaves VBASICA and returns to DOS.

REMARKS:

All files are closed before exiting VBASICA.

TAB Function

FORMAT:

```
TAB(I)
```

PURPOSE:

Moves the print position to I.

REMARKS:

If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. I must be from 1 to 255. You can use TAB only in PRINT and LPRINT statements.

EXAMPLE:

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$, B$
30 PRINT A$ TAB(25) B$
40 DATA "G.T.JONES", "$25.00"
```

yields

```
NAME          AMOUNT
G.T. JONES    $25.00
```

3

TAN Function

FORMAT:

TAN(X)

PURPOSE:

Returns the tangent of X, where X is in radians.

REMARKS:

If TAN overflows, VBASICA displays the "Overflow" error message. VBASICA supplies machine infinity with the appropriate sign as the result, and execution continues.

EXAMPLE:

```
10Y = Q*TAN(X)/2
```

TIME$ Function and Statement

FORMAT:

TIME$ = < **string expr** >		Sets the current time.
< **string expr** > = **TIME$**		Gets the current time.

PURPOSE:

Sets or retrieves the current time.

REMARKS:

< string expr > is a valid string literal or variable.

VBASICA fetches and assigns the current time to the string variable if TIME$ is the expression in a LET or PRINT statement.

VBASICA stores the current time if TIME$ is the target of a string assignment.

RULES:

1. If < string expr > is not a valid string, the "Type mismatch" error results.
2. For < string var > = TIME$, TIME$ returns an 8-character string in the form "hh:mm:ss" where hh is the hour (00 to 23), mm is the minutes (00 to 59), and ss is the seconds (00 to 59).
3. For TIME$ = < string expr >, < string expr > can take one of the following forms:
   a. hh sets the hour. Minutes and seconds default to 00.
   b. hh:mm sets the hour and minutes. Seconds default to 00.
   c. hh:mm:ss sets the hour, minutes, and seconds.

You can omit a leading zero from any of these values, but you must include at least one digit. For example, if you want to set the time as one half hour after midnight, you can enter TIME$ = "0:30", but not TIME$ = ":30".

If any of the values are out of range, VBASICA issues the "Illegal function call" error. VBASICA retains the previous time.

EXAMPLE:

```
TIME$ = "08:00"
Ok
PRINT TIME$
08:00:04
Ok
```

This program displays the current date and time on the 25th line of the screen and chimes on the hour:

```
10 KEY OFF:SCREEN 0:WIDTH 40:CLS
20 LOCATE 25,5
30 PRINT DATE$,,TIME$
40 SEC = VAL(MID$(TIME$,7,2))
50 IF SEC = SSEC THEN 20 ELSE SSEC = SEC
60 IF SEC = 0 THEN 1010
70 IF SEC = 30 THEN 1020
80 IF SEC < 57 THEN 20
1000 SOUND 1000,2:GOTO 20
1010 SOUND 2000,8:GOTO 20
1020 SOUND 400,4 :GOTO 20
```

TIMER ON, TIMER OFF, and TIMER STOP Statements

FORMAT:

TIMER ON

TIMER OFF

TIMER STOP

PURPOSE:

TIMER ON enables event trapping during real time.

TIMER OFF disables event trapping during real time.

TIMER STOP suspends real-time event trapping.

REMARKS:

The TIMER ON statement enables real time event trapping by an ON TIMER statement (see ON TIMER Statement). While ON TIMER enables trapping, VBASICA checks between every statement to see if the timer has reached the specified level. If it has, VBASICA executes the ON TIMER statement.

TIMER OFF disables the event trap. If an event occurs it is not remembered if you use a subsequent TIMER ON.

TIMER STOP disables the event trap, but if an event occurs, it is remembered and an ON TIMER statement is executed as soon as you enable trapping.

Also see the ON TIMER statement.

TRON/TROFF Commands

FORMAT:

TRON

TROFF

PURPOSE:

Traces the execution of program statements.

3

REMARKS:

Use TRON as an aid to debugging. The TRON statement (executed in direct or indirect mode) enables a trace flag that prints each line number of the program as that line is executed. The numbers are enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

EXAMPLE:

```
TRON
Ok
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
Ok
RUN
 [10][20][30][40] 1  10   20
 [50][60][30][40] 2  20   30
 [50][60][70]
Ok
TROFF
Ok
```

USR Function

FORMAT:

USR [< digit >] [(< argument >)]

PURPOSE:

Calls an assembly language subroutine.

REMARKS:

< digit > specifies the USR routine being called. See the DEF USR statement for rules governing < digit >. If you omit < digit >, VBASICA assumes USR0.

< argument > is the value passed to the subroutine. It may be any numeric or string expression. If a segment other than the default segment (data segment) is to be used, a DEF SEG statement must be executed prior to a USR function call. The address given in the DEF SEG statement determines the segment address of the subroutine.

For each USR function, a corresponding DEF USR statement must be executed to define the USR call offset. This offset and the currently active DEF SEG segment address determine the starting address of the subroutine.

EXAMPLE:

```
100 DEF SEG = &H8000
110 DEF USR0=0
120 X=5
130 Y = USR0 (X)
140 PRINT Y
```

The type (numeric or string) of the variable receiving the value must be consistent with the argument passed.

VAL Function

FORMAT:

VAL (< string >)

PURPOSE:

Returns the numeric value of < string > . The VAL function strips leading blanks, tabs, and linefeeds from the argument string. For example,


```
VAL("-3")
```

returns − 3.

REMARKS:

< string > must be a numeric value stored as a string.

VAL stops scanning the string for numeric characters as soon as it encounters either:

- Any nonnumeric character other than blank, tab, or linefeed.
- Any nonnumeric character (including blank, tab, or linefeed) after having found any numeric character(s).

If < string > contains no numeric characters, VAL returns 0 (zero).

See the STR$ function for details on numeric-to-string conversion.

EXAMPLES:

```
10 READ NAME$, CITY$, STATE$, ZIP$
20 IF VAL (ZIP$) <90000 OR VAL (ZIP$)>96699
   THEN PRINT NAME$ TAB (25) "OUT OF STATE"
30 IF VAL (ZIP$)>=90801 AND VAL (ZIP$)<=90815
   THEN PRINT NAME$ TAB (25) "LONG BEACH"
    .
    .
```

The following example:

```
1000 ADDRESS$="27 So. Spring St."
1010 NUMBER=VAL (ADDRESS$)
1020 PRINT NUMBER
```

yields

```
27
```

This example yields 0:

```
Print VAL ("ABC")
```


VARPTR Function

FORMAT:

x = VARPTR(< variable >)

y = VARPTR([#] < file number >)

PURPOSE:

For variables, returns the location in memory of the variable. For files, the VARPTR function returns the address of the first byte of the file control block (FCB) for the opened file.

REMARKS:

For both formats, the address returned is an integer from 0 to 65536. This number is the offset into the current segment of memory as defined by the DEF SEG statement.

The first format returns the address of the first byte of data identified with <variable>. Assign a value to <variable> prior to the VARPTR call, or you get an "Illegal function call" error. You may use any type variable (numeric, string, array element).

NOTE: Assign all simple variables before calling VARPTR for an array, because addresses of arrays change whenever a new simple variable is assigned.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to a machine language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

The second format returns the starting address of the file control block for the specified file. This is not the same as the DOS file control block. The file must be OPENed before the call to VARPTR.

<file number> is tied to a currently open file. Offsets to information in the FCB from the address returned by VARPTR are:

OFFSET	SIZE	CONTENTS	
0	1	Mode	The mode in which the file was opened: 1 Input Only 2 Output Only 4 Random I/O 16 Append Only 32 Internal Use 64 Future Use 128 Internal Use
1	38	FCB	Disk File Control Block. Refer to the *Systems Programmer's Tool Kit II, Volume II*, for contents.
39	2	CURLOC	Number of sectors read or written for sequential access. For Random access, it contains the last record number + 1 read or written.
41	1	ORNOFS	Number of bytes in sector when read or written.
42	1	NMLOFS	Number of bytes left in input buffer.
43	3	***	Reserved for future expansion.

OFFSET	SIZE	CONTENTS	
46	1	DEVICE	Device number: 0–9 Disks A: through J: 255 KYBD: 254 SCRN: 253 LPT1: 251 COM1: 250 COM2: 249 LPT2: 248 LPT3:
47	1	WIDTH	Device width.
48	1	POS	Position in buffer for PRINT.
49	1	FLAGS	Internal use during LOAD/SAVE. Not used for data files.
50	1	OUTPOS	Output position used during tab expansion.
51	128	BUFFER	Physical data buffer. Used to transfer data between DOS and VBASICA. Use this offset to examine data in sequential I/O mode.
179	2	VRECL	Variable-length record size. Default is 128. Set by length option in OPEN statement.
181	2	PHYREC	Current physical record number.
183	2	LOGREC	Current logical record number.
185	1	***	Future use.
186	2	OUTPOS	Disk file only. Output position for PRINT, INPUT, and WRITE.
188	<n>	FIELD	Actual FIELD data buffer. Size is determined by /S: switch. VRECL bytes are transferred between BUFFER and FIELD on I/O operations. Use this offset to examine file data in Random I/O mode.

EXAMPLE:

```
10 OPEN "DATA.FIL" AS #1
20 FCBADR = VARPTR(#1) 'FCBADR contains start of FCB
30 DATADR = FCBADR+188 'DATADR contains address of
   'data buffer.
40 A$ = PEEK (DATADR) 'A$ contains lst byte in
   'data buffer.
```

VARPTR$ Function

FORMAT:

VARPTR$ (< variable name >)

PURPOSE:

Returns a character form of the variable's memory address. The form is compatible for programs that might be compiled later.

3

REMARKS:

< variable name > is the name of a variable in the program.

VARPTR$ executes substrings with the DRAW and PLAY statements in programs that will later be compiled. With programs that will not be later compiled, the standard syntax of the PLAY and DRAW statements is sufficient to produce desired effects.

You must assign a value to the variable before calling VARPTR$. Otherwise, an "Illegal function call" error results. Variables are defined by executing any reference to the variable.

VARPTR$ returns a three-byte string in the form:

byte 0 = type
byte 1 = low byte of address
byte 2 = high byte of address

The individual parts of the string are not considered characters.

NOTE: Because array addresses, string addresses and file data blocks change whenever a new variable is assigned, it is unsafe to save the result of a VARPTR function in a variable. Execute VARPTR before each use of the result.

EXAMPLE:

```
10 PLAY "X" + VARPTR$(A$)
```

Uses the subcommand X (execute), plus the contents of A$, as the string expression in the PLAY statement.

VIEW Statement

FORMAT:

VIEW [[SCREEN] [(Vx1,Vy1)-(Vx2,Vy2) [,[< color >][,[< border >]]]]]

PURPOSE:

Defines the screen limits for graphics activity, in Graphics mode only.

REMARKS:

VIEW defines a "Physical Viewport" limit from Vx1, Vy1 (upper left x,y coordinates) to Vx2, Vy2 (lower right x,y coordinates). The x and y coordinates must be within the physical bounds of the screen. The physical viewport defines the rectangle within the screen into which you can map graphics.

RUN, and RUN, SCREEN, and VIEW with no arguments, define the entire screen as the viewport.

With the < color > attribute, you can fill the view area with a color. If you omit color, VBASICA does not fill the view area.

With the < border > attribute, you can draw a line surrounding the viewport if space for a border is available. If you omit < border >, VBASICA draws no border.

The [SCREEN] option dictates that the x and y coordinates are absolute to the screen, not relative to the border of the physical viewport, and VBASICA plots only graphics within the viewport.

Out-of-range coordinates are clipped.

EXAMPLE:

For the following form, all points plotted are relative to the viewport. That is, Vx1 and Vy1 are added to the x and y coordinates before plotting the point on the screen.


```
VIEW (Vx1,Vy1)-(Vx2,Vy2)
```

If the following command is executed, then the point set down by the statement PSET (0,0),3 will be at the physical screen location 10,10.

```
VIEW (10,10)-(200,100)
```

For the following form, all coordinates are screen absolute rather than viewport relative:

```
VIEW SCREEN (Vx1,Vy1)-(Vx2,Vy2)
```

If VBASICA executes the following, then the point set down by the statement PSET (0,0),3 will not appear because 0,0 is outside the viewport. PSET (10,10),3 is within the viewport, and places the point in the upper left corner of the viewport.

```
VIEW SCREEN (10,10)-(200,100)
```

Many VIEW statements can be executed. If the newly described viewport is not wholly within the previous viewport, the screen can be reinitialized with the VIEW statement. Then the new viewport can be stated. If the new viewport is entirely within the previous one, as in the following example, the intermediate VIEW statement isn't necessary.

This example opens three viewports, each smaller than the previous one. In each case, a line defined to go beyond the borders is programmed, but appears only within the viewport border.

```
280 VIEW:REM ** Make the viewport the entire screen
300 VIEW (10,10) - (300,180),,1
320 CLS
340 LINE (0,0) -(310,190),1
360 LOCATE 1,11: PRINT "A big viewport"
380 VIEW SCREEN (50,50)-(250,150),,1
400 CLS:REM**Note, CLS clears only viewport
420 LINE (300,0)-(0,199),1
440 LOCATE 9,9: PRINT "A medium viewport"
460 VIEW SCREEN (80,80)-(200,125),,1
480 CLS
500 CIRCLE (150,100),20,1
520 LOCATE 11,9: PRINT "A small viewport"
```

VIEW PRINT Statement

FORMAT:

VIEW PRINT [< top screen line > TO < bottom screen line >]

PURPOSE:

Sets the boundaries of the screen text window.

3

REMARKS:

VIEW PRINT without top and bottom line parameters initializes the whole screen area as the text window.

Statements and functions that operate within the defined text window include CLS, LOCATE, PRINT, and SCREEN.

The Screen Editor limits functions such as scroll and cursor movement to the text window.

Also see the VIEW statement.

WAIT Statement

FORMAT:

WAIT <**port**>,<**and byte**> [,<**xor byte**>]

PURPOSE:

Suspends program execution while monitoring the status of a machine port.

REMARKS:

<port> is a numeric expression returning an integer from 0 to 65535.

<and byte> is a numeric expression returning an integer from 0 to 255 and matches a byte coming in from the <port>.

<xor byte> is a numeric expression returning an integer from 0 to 255 and checks a byte coming in from the <port>.

The WAIT statement suspends execution until a specified machine input port develops a specified bit pattern. The data read at the port is XOR'ed with the integer expression XOR byte and then AND'ed with the AND byte. If the result is zero, VBASICA loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement.

WARNING: It is possible to enter an infinite loop with the WAIT statement. CTRL-C exits the loop.

WHILE...WEND Statement

FORMAT:

```
WHILE < expr >
      .
      .
[ < loop statements > ]
      .
      .
WEND
```


PURPOSE:

Executes a series of statements in a loop as long as a given condition is true.

REMARKS:

If < expr > is not zero (that is, true), the loop statements are executed until the WEND statement is encountered. VBASICA then returns to the WHILE statement and checks < expr > . If < expr > is still true, the process is repeated. If < expr > is not true, execution resumes at the statement after the WEND statement. WHILE/WEND loops can be nested to any level. Each WEND matches the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error; an unmatched WEND statement causes a "WEND without WHILE" error.

EXAMPLE:

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115     FLIPS=0
120     FOR I=1 TO J-1
130         IF A$(I)>A$(I+1) THEN
                SWAP A$(I), A$(I+1):FLIPS=1
140     NEXT I
150 WEND
```

WIDTH Statement

FORMAT:

WIDTH < **size** >

WIDTH < **file no.** > , < **size** >

WIDTH < **dev** > , < **size** >

PURPOSE:

Sets the printed line width in number of characters for the screen and line printer.

REMARKS:

< size > is a valid numeric expression returning an integer result from 0 to 255. This value is the new width.

< file no. > is a valid numeric expression returning an integer. This value is the number of the file OPENed.

< dev > is a valid string expression returning the device identifier.

Depending on the device specified, the following actions are possible:

```
WIDTH <size>
WIDTH "SCRN:", <size>
```

These commands set the screen width. VBASICA allows only 40- or 80-column width.

NOTE: Changing the screen width clears the screen. Screen mode 1 is always 40 columns wide, and Screen mode 2 is always 80 columns. If you are in either of these modes and ask to change the width, the mode changes to the appropriate one of these two modes.

```
WIDTH "LPT1:",<size>
```

Used as a deferred width assignment for the line printer. This form of WIDTH stores the new width value without changing the current width setting. A subsequent OPEN "LPT1:" FOR OUTPUT AS < number > uses this value for width while the file is open.


```
WIDTH <file no.>,<size>
```

If the file is open to LPT1:, the line printer's width is immediately changed to the new size specified. This allows the width to be changed while the file is open. This form of WIDTH affects only LPT1:.

RULES:

1. Valid widths for the screen are 40 and 80. The valid width for the line printer is 1 to 255. Any value entered outside these ranges results in the "Illegal function call" error. VBASICA retains the previous value.
2. Width has no effect on the keyboard (KYBD:).
3. Maximum Epson line printer width is 80. WIDTH, however, does not complain about values between 80 and 255.
4. Specifying WIDTH 255 for the line printer (LPT1:) disables line folding.

EXAMPLE:

```
10 WIDTH "LPT1:",75
20 OPEN "LPT1:" FOR OUTPUT AS #1
        .
        .
        .
6020 WIDTH #1,40
```

In the preceding example, line 10 stores a line printer width of 75 characters per line. Line 20 opens file #1 to the line printer and sets the width to 75 for subsequent PRINT #1,... statements. Line 6020 changes the current line printer width to 40 characters per line.

WINDOW Statement

FORMAT:

WINDOW [[SCREEN] (Wx1,Wy1)-(Wx2,Wy2)]

PURPOSE:

Defines the logical dimensions of the current viewport, for Graphics mode only.

REMARKS:

(Wx1,Wy1)-(Wx2,Wy2) are the world coordinates you specify to define the coordinates of the lower left and upper right screen border.

SCREEN inverts the y-axis of the world coordinates so that screen coordinates follow the traditional Cartesian system: x increases left to right, and y decreases top to bottom.

With WINDOW, you can redefine the screen border coordinates. You can also draw lines, graphs, or objects in space not bounded by the physical dimensions of the screen with world coordinates. When you redefine the screen, you can draw graphics within a customized mapping system.

VBASICA converts world coordinates into physical coordinates for subsequent display within the current viewport. To make this transformation from world space to the physical space of the viewing surface (screen), you must know what portion of the (floating-point) world coordinate space contains the information to be displayed. This rectangular region in world coordinate space is a window.

RUN or WINDOW with no arguments disables window transformation. The WINDOW SCREEN variant inverts the normal Cartesian direction of the y coordinate.

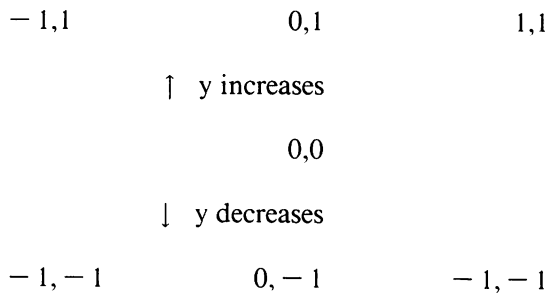
For example, in the default, a section of the screen appears as:

```
0,0          50,0          100,0
          ↓ y increases
             100,0
0,100        50,100        100,100
```

Now execute the following:

```
WINDOW (-1,-1)-(1,1)
```

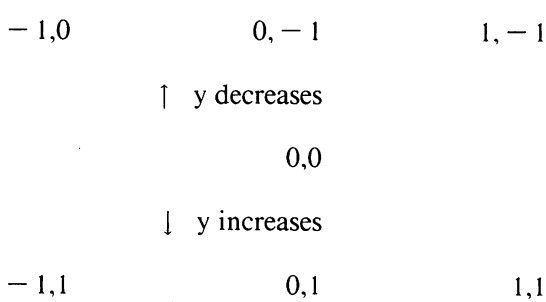
and the screen appears as:



If the variant:

```
WINDOW SCREEN (-1,-1)-(1,1)
```

is executed then the screen appears as:



The following example illustrates two lines with the same endpoint coordinates. The first is drawn on the default screen, and the second is on a redefined window.

```
200 LINE (100,100) - (150,150), 1
220 LOCATE 2,20:PRINT "The line on the default screen"
240 WINDOW SCREEN (100,100) - (200,200)
260 LINE (100,100) - (150,150), 1
280 LOCATE 8, 18:PRINT "& the same line on
    a redefined window"
```


WRITE Statement

FORMAT:

WRITE[< expr list >]

PURPOSE:

Displays data on the screen.

REMARKS:

< expr list > is a list of numeric and/or string expressions. Commas separate the expressions. If < expr list > is omitted, a blank line is output. If < expr list > is included, the values of the expressions are displayed on your screen.

The output items are separated by commas. Strings are delimited by quotation marks. After the last item in the list appears, VBASICA inserts a carriage return/linefeed.

WRITE outputs numeric values using the same format as the PRINT statement.

EXAMPLE:

```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
RUN
 80,90,"THAT'S ALL"
Ok
```

WRITE# Statement

FORMAT:

WRITE# < filenum > , < expr list >

PURPOSE:

Writes data to a sequential file.

REMARKS:

< filenum > is the number under which a file was opened.

The expressions in < expr list > are string or numeric expressions, separated by commas.

Unlike PRINT#, WRITE# inserts commas between items as they are written to disk, and delimits strings with quotation marks. You do not need to put explicit delimiters in the list. A carriage return/linefeed sequence is inserted after the last item is written to disk.

Assume that A$ is "CAMERA" and B$ is "93604-1". The statement:

```
WRITE#1,A$,B$
```

writes this to disk:

```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as:

```
INPUT#1,A$,B$
```

inputs "CAMERA" to A$ and "93604-1" to B$.

4

VBASICA and Communications

This chapter describes the VBASICA statements required to support asynchronous serial communication with other computers and peripherals, through the RS-232-C ports.

4.1 Communication I/O

Because you open the communications port as a file, all Input/output statements valid for disk files are valid for COM.

COM sequential input statements are the same as those for disk files: INPUT # < file number >, LINE INPUT # < file number >, and the INPUT$ function.

COM sequential output statements are the same as those for disk: PRINT # < file number > and PRINT # < file number > USING.

Refer to INPUT and PRINT for details of coding syntax and usage.

GET and PUT are only slightly different for COM files. See the GET and PUT statements for COM.

The TTY program that follows enables your VBASICA computer to be used as a conventional terminal. In addition to full-duplex communication with a host, the TTY program allows data to be downloaded to a file. Conversely, a file can be up-loaded (transmitted) to another machine.

In addition to demonstrating the elements of asynchronous communication, this program is useful in transferring VBASICA programs and data to and from your computer.

NOTE: The TTY program is set up to communicate using XON and XOFF. You may want to modify it for your environment.

4.2 The TTY Program

```
10 SCREEN 0,0:WIDTH 80
15 KEY OFF:CLS:CLOSE
20 DEFINT A-Z
25 LOCATE 25,1
30 PRINT STRING$(60," ")
40 FALSE=0:TRUE= NOT FALSE
50 MENU=5 ' Value of MENU key (CTRL-E)
60 XOFF$=CHR$(19):XON$=CHR$(17)

100 LOCATE 25,1:PRINT "Async TTY Program ";
110 LOCATE 1,1:LINE INPUT "Speed? ";SPEED$
120 COMFIL$="COM1:"+SPEED$+",E,7"
130 OPEN COMFIL$ AS #1

200 PAUSE=FALSE
210 A$=INKEY$: IF A$="" THEN 230
220 IF ASC(A$)=MENU THEN 300 ELSE PRINT #1,A$;
230 IF EOF(1) THEN 210
240 IF LOC(1)>128 THEN PAUSE=TRUE: PRINT #1,XOFF$;
250 A$=INPUT$(LOC(1),#1)
260 PRINT A$;:IF LOC(1)>0 THEN 240
270 IF PAUSE THEN PAUSE=FALSE:PRINT #1,XON$;
280 GOTO 210

300 LOCATE 1,1:PRINT STRING$(30," "):LOCATE 1,1
310 LINE INPUT"File? ";DSKFIL$

400 LOCATE 1,1:PRINT STRING$(30," "):LOCATE 1,1
410 LINE INPUT"(T)ransmit or (R)eceive? ";TXRX$
420 IF TXRX$="T" THEN OPEN DSKFIL$ FOR INPUT
    AS #2:GOTO 1000
430 OPEN DSKFIL$ FOR OUTPUT AS #2
440 PRINT #1,CHR$(13);
```

```
500 IF EOF(1) THEN GOSUB 600
510 IF LOC(1)>128 THEN PAUSE=TRUE: PRINT #1,XOFF$;
520 A$=INPUT$(LOC(1),#1)
530 PRINT #2,A$;:IF LOC(1)>0 THEN 510
540 IF PAUSE THEN PAUSE=FALSE:PRINT #1,XON$;
550 GOTO 500

600 FOR I=1 TO 5000
610 IF NOT EOF(1) THEN I=9999
620 NEXT I
630 IF I>9999 THEN RETURN
640 CLOSE #2:CLS:LOCATE 25,10:PRINT
    "* Download complete *";
650 GOTO 200

1000 WHILE NOT EOF(2)
1010 A$=INPUT$(1,#2)
1020 PRINT #1,A$;
1030 WEND
1040 PRINT #1,CHR$(26); 'CTRL-Z to make close file.
1050 CLOSE #2:CLS:LOCATE 25,10:PRINT
     "** Upload complete **";
1060 GOTO 200

9999 CLOSE:KEY ON
```


4.3 Notes on the TTY Program

LINE NUMBER	COMMENTS
10	Sets the screen to Alpha mode and sets the width to 80.
15	Turns off the soft key display, clears the screen, and ensures all files are closed.
	NOTE: Asynchronous implies character I/O, as opposed to line or block I/O. Therefore, terminate all PRINTs, either to the COM file or to the screen, with a semicolon (;). This action suppresses the carriage return/line feed normally issued at the end of a PRINT statement.

LINE NUMBER	COMMENTS
20	Defines all numeric variables as INTEGER. Primarily for the subroutine at 600–620.
25–30	Clears the 25th line starting at column 1.
40	Defines Boolean TRUE and FALSE.
50	Defines the ASCII (ASC) value of the MENU key.
60	Defines the ASCII XON, XOFF characters.
100–130	Prints program ID and asks for baud rate (speed). Opens communications to file number 1, even parity, 7 data bits. Programmer exercise: Modify this section to check for valid baud rates.
200–280	Performs full-duplex I/O between the video screen and the device connected to the RS-232-C connector as follows: 1. Read a character from the keyboard into A$. INKEY$ returns a null string if no character is waiting. 2. If no character is waiting, then check if any characters are being received. If a character is waiting at the keyboard, and: — If the character is the MENU key, the user is ready to download a file, so get the filename. — If character (A$) is not the MENU key, send it by writing to the communication file (PRINT #1...). 3. At 230 see if any characters are waiting in COM buffer. If not, then go back and check keyboard. 4. At 240, if more than 128 characters are waiting, then set PAUSE flag saying we are suspending input. Send XOFF to host, stopping further transmission. 5. At 250–260, read and display contents of COM buffer on screen until empty. Continue to monitor size of COM buffer (in 240). Suspend transmission if the program falls behind. 6. Finally, resume host transmission by sending XON only if suspended by previous XOFF. Repeat process until MENU key is struck.
300–310	Get disk file name we are downloading to. Open file to tie number 2.
400–420	Asks if file named is to be transmitted (uploaded) or received (downloaded).

LINE NUMBER	COMMENTS
430	Sends a carriage return to the host to begin the download. This program assumes that the last command sent to the host was to begin such a transfer and was missing only the terminating carriage return. If a DEC system is the host, then such a command might be the following: **COPY TTY: = MANUAL.MEM < MENU key >** where the MENU key was struck instead of Return.
500	When no more characters are being received (LOC(x) returns 0), then perform a time-out routine (see line 600).
510	Again, if more than 128 characters are waiting, signal a pause and send XOFF to the host while up.
520–530	Read all characters in COM queue (LOC(x)) and write them to disk (PRINT #2..) until caught up.
540–550	If a pause was issued, restart host by sending XON and clear the pause flag. Continue process until no characters are received for a predetermined time.
600–650	This is the time-out subroutine. The FOR loop count was determined by experimentation. In short, if no character is received from the host for 17–20 seconds, then transmission is assumed complete. If any character is received during this time (line 610), then I is set well above FOR loop range to exit loop and then return to caller. If host transmission is complete, close the disk file and return to being a terminal.
1000–1060	Transmit routine. Until end of disk file do the following: Read one character into A$ with INPUT$ statement. Send character to COM device in 1020. Send a CTRL-Z at end of file in 1040 in case receiving device needs one to close its file. Finally, in lines 1050 and 1060, close our disk file, print completion message and go back to conversion mode in line 200.
9999	Presently not executed. As an exercise, add some lines to the routine 400–420 to exit the program via line 9999. This line closes the COM file left open and restores the soft key display.

4.4 The COM I/O Functions

The most difficult aspect of asynchronous communication is processing characters as fast as they are received. At rates above 2400 bps, character transmission must be suspended from the host long enough to catch up. This suspension can be done by sending XOFF (CTRL-S) to the host and XON (CTRL-Q) when ready to resume.

VBASICA provides three functions to help determine when an "overrun" condition is imminent:

LOC(x) Returns the number of characters in the input queue waiting to be read. The input queue can hold more than 255 characters (determined by the /C: switch). If there are more than 255 characters in the queue, LOC(x) returns 255. Because a string is limited to 255 characters, this practical limit alleviates the need for the programmer to test for string size before reading data into it. If fewer than 255 characters remain in the queue, LOC(x) returns the actual count.

LOF(x) Returns the amount of free space in the input queue—that is, /C: < size > − LOC(x). You can use LOF to detect when the input queue is getting full. LOC is adequate for this purpose, as shown in the programming example.

EOF(x) If true (− 1), indicates that the input queue is empty. Returns false (0) if any characters are waiting to be read.

These errors can occur:

1. "Communication Buffer Overflow" occurs if a read is attempted after the input queue is full (that is, LOC(x) returns 0).
2. "Device I/O Error" occurs if any of the following line conditions are detected on receive: Overrun Error (OE), Framing Error (FE), or Break Interrupt (BI). The error is reset by subsequent inputs but the character causing the error is lost.
3. "Device Fault" occurs if data set ready (DSR) is lost during I/O.

Error Messages

VBASICA provides the following error messages. The error codes and messages described in this appendix can appear when you are using the VBASICA Interpreter. Each message is explained and corrective measures are suggested, when appropriate.

CODE NUMBER	MESSAGE
1	**NEXT without FOR** A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
2	**Syntax error** A line contains an incorrect sequence of characters, such as unmatched parentheses, misspelled command or statement, incorrect punctuation, and so on.
3	**Return without GOSUB** A RETURN statement appears without a previous, unmatched GOSUB statement.
4	**Out of data** VBASICA executes a READ statement when no remaining DATA statements exist that contain unread data.
5	**Illegal function call** An out-of-range parameter is passed to a math or string function. An FC error can also occur as the result of: ▶ A negative or unreasonably large subscript ▶ A negative or zero argument with LOG ▶ A negative argument to SQR ▶ A negative mantissa with a noninteger exponent ▶ A call to a USR function for which the starting address was not given ▶ An incorrect argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO

CODE NUMBER	MESSAGE
6	**Overflow** The result of a calculation is too large to be represented in VBASICA number format. If underflow occurs, the result is zero and execution continues without an error.
7	**Out of memory** A program is too large, has too many FOR loops or GOSUBs, has too many variables, or contains expressions that are too complicated.
8	**Undefined line** A line reference in a GOTO, GOSUB, IF...THEN...ELSE, or DELETE refers to a nonexistent line.
9	**Subscript out of range** An array element is referenced with a subscript outside the dimensions of the array or with the wrong number of subscripts.
10	**Redimensioned array** Two DIM statements are given for the same array, or a DIM statement is given for an array when the default dimension is 10.
11	**Division by zero** An expression contains a division by zero or the operation of involution raises zero to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution. In both cases, execution continues.
12	**Illegal direct** You have entered a statement that is illegal in direct mode.
13	**Type mismatch** A string variable name is assigned a numeric value (or vice versa), or a function that expects a numeric argument is given a string argument (or vice versa).
14	**Out of string space** String variables caused VBASICA to exceed the remaining free memory. VBASICA allocates string space dynamically until it runs out of memory.
15	**String too long** You have tried to create a string more than 255 characters long.
16	**String formula too complex** A string expression is too long or too complex. The expression must be broken into smaller expressions.

CODE NUMBER	MESSAGE
17	**Can't continue** You have tried to continue a program that: ▶ Has halted due to an error ▶ Modified during a break in execution ▶ Does not exist
18	**Undefined user function** A USR function is called before the function definition (DEF statement) is given.
19	**No RESUME** You entered an error-trapping routine that lacks a RESUME statement.
20	**RESUME without error** VBASICA encounters a RESUME statement before it enters an error-trapping routine.
21	**Unprintable error** An error message is not available for the error condition that exists. An error with an undefined error code usually causes this error.
22	**Missing operand** An expression contains an operator without a following operand.
23	**Line buffer overflow** You tried to input a line with too many characters.
24	**Device Timeout** VBASICA does not receive information from an I/O device within a predetermined amount of time.
25	**Device I/O Error** Fault status is returned from the parallel and serial devices. Usually indicates a hardware error in the printer or serial communications channel.
26	**FOR without NEXT** A FOR does not have a matching NEXT.
29	**WHILE without WEND** A WHILE statement does not have a matching WEND.
30	**WEND without WHILE** A WEND does not have a matching WHILE.
50	**Field overflow** A FIELD statement tried to allocate more bytes than were specified for the record length of a random file.

CODE NUMBER	MESSAGE
51	**Internal error** An internal malfunction has occurred in VBASICA. Report to your dealer the conditions under which the message appeared.
52	**Bad file number** A statement or command references a file with a file number that is not OPEN, or is out of the range of file numbers specified at initialization.
53	**File not found** A LOAD, KILL, or OPEN statement references a disk file that does not exist.
54	**Bad file mode** You have tried to use PUT, GET, or LOF with a sequential file, to LOAD a random file, or to execute an OPEN with a file mode other than I, O, or R.
55	**File already open** VBASICA issues a sequential output mode OPEN for a file already open, or a KILL is given for an open file.
57	**I/O error** An I/O error occurs on a device I/O operation. This error is fatal; the operating system cannot recover from the error. Formerly was Disk I/O error.
58	**File already exists** The filename specified in a NAME statement is identical to a filename already in use on the disk.
61	**Disk full** All disk storage space is in use.
62	**Input past end** VBASICA executes an INPUT statement for a null (empty) file, or after all the data in the file was INPUT. To avoid this error, use the EOF function to detect end-of-file.
63	**Bad record number** The record number in a PUT or GET statement is greater than the maximum allowed (32,767) or equal to zero.
64	**Bad file name** An illegal form is used for the filename in a LOAD, SAVE, KILL, or OPEN statement, for example, a filename with too many characters.
66	**Direct statement in file** VBASICA encountered a direct statement in the file while LOADing an ASCII-format file. VBASICA terminates the LOAD.

CODE NUMBER	MESSAGE
67	**Too many files** You tried to create a new file (using SAVE or OPEN) when all 255 directory entries are full.
68	**Device Unavailable** An attempt is made to open a file to a nonexistent device. Hardware may not exist to support the device, such as LPT2: or LPT3:, or you may have disabled it. This error occurs if VBASICA executes an OPEN "COM1:... statement but you disable RS-232-S with the /C:0 switch directive on the command line.
69	**Communication Buffer Overflow** VBASICA executes a communication input statement but the input queue is already full. Use an ON ERROR GOTO statement to retry the input when this condition occurs. Subsequent inputs attempt to clear this fault unless characters continue to be received faster than the program can process them. In this case several options are available: 1. Increase the size of the COM receive buffer with the /C: switch. 2. Implement a hand-shaking protocol with the host/satellite such as XON/XOFF, as demonstrated in the TTY programming example, to turn transmit off long enough to catch up. 3. Use a lower baud rate for transmit and receive.
70	**Disk Write Protect** One of the three hard disk errors returned from the diskette controller. This error occurs when you try to write to a write-protected diskette. Use an ON ERROR GOTO statement to detect this situation and request user action.
71	**Disk Not Ready** The diskette drive door is open or a diskette is not in the drive. Recover with an ON ERROR GOTO statement.
72	**Disk Media Error** Occurs when the FDC controller detects a hardware or media fault. This error usually indicates harmed media. Copy existing files to a new diskette and reformat the damaged diskette. FORMAT flags the bad tracks and places them in a file badtrack. The remainder of the diskette is now usable.

Extended Codes

Certain keys or combinations of keys cannot return a value within the ASCII code range. These keys are remapped to generate an extended code when VBASICA executes an INKEY$ statement. The codes returned by the INKEY$ statement consist of an ASCII null (00) as the first part of the two-byte string. If a two-byte string is received by INKEY$, then check the second key value to determine the key pressed. The ASCII code in decimal and the associated key(s) are shown below:

SECOND CODE	MEANING
15	Shift Tab, ←
59–68	Function keys 1 through 10 (when not used as soft keys)
71	Home
72	Cursor Up
75	Cursor Left
77	Cursor Right
80	Cursor Down
82	Insert
83	Delete
84–93	Shifted function keys 1 through
94–103	CTRL function keys 1 through
119	CTRL-Home

NOTE: This appendix will be enh :d with future releases of this product.

ASCII Character Codes

ASCII Value		Screen	Code	
Decimal	Hex	Character	Name	Keystroke
000	00	(null)	NUL	—
001	01	☺	SOH	CTRL-A
002	02	☻	STX	CTRL-B
003	03	♥	ETX	CTRL-C
004	04	♦	EOT	CTRL-D
005	05	♣	ENQ	CTRL-E
006	06	♠	ACK	CTRL-F
007	07	(beep)	BEL	CTRL-G
008	08	◘	BS	CTRL-H
009	09	(tab)	HT	CTRL-I
010	0A	(line feed)	LF	CTRL-J
011	0B	(home)	VT	CTRL-K
012	0C	(form feed)	FF	CTRL-L
013	0D	(carriage return)	CR	CTRL-M
014	0E	♫	SO	CTRL-N
015	0F	☼	SI	CTRL-O
016	10	►	DLE	CTRL-P
017	11	◄	DC1	CTRL-Q
018	12	↕	DC2	CTRL-R
019	13	!!	DC3	CTRL-S
020	14	¶	DC4	CTRL-T
021	15	§	NAK	CTRL-U
022	16	▬	SYN	CTRL-V
023	17	↨	ETB	CTRL-W
024	18	↑	CAN	CTRL-X
025	19	↓	EM	CTRL-Y
026	1A	→	SUB	CTRL-Z
027	1B	←	ESC	Escape Key
028	1C	(cursor right)	FS	
029	1D	(cursor left)	GS	
030	1E	(cursor up)	RS	
031	1F	(cursor down)	US	

ASCII Value Decimal	Hex	Screen Character	ASCII Value Decimal	Hex	Screen Character
032	20	(space)	064	40	@
033	21	!	065	41	A
034	22	"	066	42	B
035	23	#	067	43	C
036	24	$	068	44	D
037	25	%	069	45	E
038	26	&	070	46	F
039	27	'	071	47	G
040	28	(	072	48	H
041	29	)	073	49	I
042	2A	*	074	4A	J
043	2B	+	075	4B	K
044	2C	,	076	4C	L
045	2D	-	077	4D	M
046	2E	.	078	4E	N
047	2F	/	079	4F	O
048	30	0	080	50	P
049	31	1	081	51	Q
050	32	2	082	52	R
051	33	3	083	53	S
052	34	4	084	54	T
053	35	5	085	55	U
054	36	6	086	56	V
055	37	7	087	57	W
056	38	8	088	58	X
057	39	9	089	59	Y
058	3A	:	090	5A	Z
059	3B	;	091	5B	[
060	3C	<	092	5C	\
061	3D	=	093	5D	]
062	3E	>	094	5E	^
063	3F	?	095	5F	_

ASCII Value Decimal	Hex	Screen Character	ASCII Value Decimal	Hex	Screen Character
096	60	`	128	80	Ç
097	61	a	129	81	ü
098	62	b	130	82	é
099	63	c	131	83	â
100	64	d	132	84	ä
101	65	e	133	85	à
102	66	f	134	86	å
103	67	g	135	87	ç
104	68	h	136	88	ê
105	69	i	137	89	ë
106	6A	j	138	8A	è
107	6B	k	139	8B	ï
108	6C	l	140	8C	î
109	6D	m	141	8D	ì
110	6E	n	142	8E	Ä
111	6F	o	143	8F	Å
112	70	p	144	90	É
113	71	q	145	91	æ
114	72	r	146	92	Æ
115	73	s	147	93	ô
116	74	t	148	94	ö
117	75	u	149	95	ò
118	76	v	150	96	û
119	77	w	151	97	ù
120	78	x	152	98	ÿ
121	79	y	153	99	Ö
122	7A	z	154	9A	Ü
123	7B	{	155	9B	¢
124	7C	|	156	9C	£
125	7D	}	157	9D	¥
126	7E	~	158	9E	Pt
127	7F	⌂	159	9F	ƒ

ASCII Value Decimal	Hex	Screen Character	ASCII Value Decimal	Hex	Screen Character
160	A0	á	192	C0	└
161	A1	í	193	C1	┴
162	A2	ó	194	C2	┬
163	A3	ú	195	C3	├
164	A4	ñ	196	C4	─
165	A5	Ñ	197	C5	┼
166	A6	ª	198	C6	╞
167	A7	º	199	C7	╟
168	A8	¿	200	C8	╚
169	A9	⌐	201	C9	╔
170	AA	¬	202	CA	╩
171	AB	½	203	CB	╦
172	AC	¼	204	CC	╠
173	AD	¡	205	CD	═
174	AE	«	206	CE	╬
175	AF	»	207	CF	╧
176	B0	▒	208	D0	╨
177	B1	▒	209	D1	╤
178	B2	▓	210	D2	╥
179	B3	│	211	D3	╙
180	B4	┤	212	D4	╘
181	B5	╡	213	D5	╒
182	B6	╢	214	D6	╓
183	B7	╖	215	D7	╫
184	B8	╕	216	D8	╪
185	B9	╣	217	D9	┘
186	BA	║	218	DA	┌
187	BB	╗	219	DB	■
188	BC	╝	220	DC	▄
189	BD	╜	221	DD	▌
190	BE	╛	222	DE	▐
191	BF	┐	223	DF	▀

C

ASCII Value		Screen
Decimal	Hex	Character
224	E0	α
225	E1	β
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	μ
231	E7	τ
232	E8	Φ
233	E9	Θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	∅
238	EE	∈
239	EF	∩
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	∙
250	FA	·
251	FB	√
252	FC	ⁿ
253	FD	²
254	FE	■
255	FF	(blank 'FF')

Mathematical Functions

Functions not intrinsic to VBASICA are calculated as follows.

FUNCTION	VBASICA EQUIVALENT
Secant	SEC(X) = 1/COS(X)
Cosecant	CSC(X) = 1/SIN(X)
Cotangent	COT(X) = 1/TAN(X)
Inverse sine	ARCSIN(X) = ATN(X/SQR(− X*X + 1))
Inverse cosine	ARCCOS(X) = − ATN(X/SQR(− X*X + 1)) + 1.5708
Inverse secant	ARCSEC(X) = ATN(X/SQR(X*X − 1)) + SGN(SGN(X) − 1)*1.5708
Inverse cosecant	ARCCSC(X) = ATN(X/SQR(X*X − 1)) + (SGN(X) − 1)*1.5708
Inverse cotangent	ARCCOT(X) = ATN(X) + 1.5708
Hyperbolic sine	SINH(X) = (EXP(X) − EXP(− X))/2
Hyperbolic cosine	COSH(X) = (EXP(X) + EXP(− X))/2
Hyperbolic tangent	TANH(X) = EXP(− X)/EXP(X) + EXP(− X))*2 + 1
Hyperbolic secant	SECH(X) = 2/(EXP(X) + EXP(− X))
Hyperbolic cosecant	CSCH(X) = 2/(EXP(X) − EXP(− X))
Hyperbolic cotangent	COTH(X) = EXP(− X)/(EXP(X) − EXP(− X))*2 + 1
Inverse hyperbolic sine	ARCSINH(X) = LOG(X + SQR(X*X + 1))
Inverse hyperbolic cosine	ARCCOSH(X) = LOG(X + SQR(X*X − 1))
Inverse hyperbolic tangent	ARCTANH(X) = LOG((1 + X)/(1 − X))/2
Inverse hyperbolic secant	ARCSECH(X) = LOG((SQR(− X*X + 1) + 1)/X)
Inverse hyperbolic cosecant	ARCCSCH(X) = LOG((SGN(X)*SQR(X*X + 1) + 1)/X
Inverse hyperbolic cotangent	ARCCOTH(X) = LOG((X + 1)/(X − 1))/2

Keyboard Scan-Codes

These are the scan-codes for an American keyboard.

KEY ID	HEX SCAN CODE	DECIMAL SCAN CODE	KEY ID	HEX SCAN CODE	DECIMAL SCAN CODE
ESC	01	01	|	2B	43
1!	02	02	Z	2C	44
2@	03	03	X	2D	45
3#	04	04	C	2E	46
4$	05	05	V	2F	47
5%	06	06	B	30	48
6^	07	07	N	31	49
7&	08	08	M	32	50
8*	09	09	<,	33	51
9(	0A	10	>.	34	52
0)	0B	11	?/	35	53
-_	0C	12	Shift(Right)	36	54
= +	0D	13	PRTSC*	37	55
Backspace	0E	14	ALT	38	56
Tab	0F	15	Space	39	57
Q	10	16	Caps Lock	3A	58
W	11	17	F1	3B	59
E	12	18	F2	3C	60
R	13	19	F3	3D	61
T	14	20	F4	3E	62
Y	15	21	F5	3F	63
U	16	22	F6	40	64
I	17	23	F7	41	65
O	18	24	F8	42	66
P	19	25	F9	43	67
[	1A	26	F10	44	68
]	1B	27	Num Lock	45	69
Return/Enter	1C	28	Scroll Lock	46	70
CTRL	1D	29	7 home	47	71
A	1E	30	8 Up	48	72
S	1F	31	9 Pg Up	49	73
D	20	32	-	4A	74
F	21	33	4 left	4B	75
G	22	34	5	4C	76
H	23	35	6 Right	4D	77
J	24	36	+	4E	78
K	25	37	1 End	4F	79
L	26	38	2 Down	50	80
;:	27	39	3 Pg Dn	51	81
~`	29	41	.DEL	53	83
Shift(Left)	2A	42			

Converting Programs to VBASICA

If you have programs written in a BASIC other than VBASICA, some minor adjustments may be necessary before running them. This appendix describes specific things to look for when converting BASIC programs to VBASICA.

String Dimensions

Delete all statements that declare the length of strings. A statement such as DIM A$(I,J), which dimensions a one-dimensional string array for J elements of length I, should be converted to the VBASICA statement DIM A$(J).

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, the operator used in VBASICA for string concatenation.

In VBASICA, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms used by other BASICs, such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

OTHER BASIC	VBASICA
X$ = A$(I)	X$ = MID$(A$,I,1)
X$ = A$(I,J)	X$ = MID$(A$,I,J-I + 1)

If the substring reference is on the left side of an assignment and you use X$ to replace characters in A$, convert as follows:

OTHER BASIC	VBASICA
A$(I) = X$	MID$(A$,1,1) = X$
A$(I,J9) = X$	MID$(A$,I,J-I + 1) = X$

Multiple Assignments

Some BASICs allow statements of the form:

```
10 LET B=C=0
```

to set B and C equal to zero. If this statement appears in a VBASICA program, VBASICA interprets the second equal sign as a logical operator, and sets B equal to −1 if C is equal to 0. To make sure that this statement is interpreted correctly, convert it to two assignment statements:

```
10 C=0:B=0
```

Multiple Statements

Some BASICs use a backslash (\) to separate multiple statements on a line. With VBASICA, be sure all statements on a line are separated by a colon (:).

MAT Functions

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to execute properly.

VBASICA Disk I/O

This appendix describes disk I/O procedures. If you are new to VBASICA or you are getting disk-related errors, read through these procedures and program examples to make sure you're using all the disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to your operating system's requirements.

G.1 Program File Commands

Here is a review of the commands and statements used in program file manipulation.

```
SAVE<filename>[,A]
```

Writes the program in memory to a disk file. The A option writes the program as a series of ASCII characters; otherwise, VBASICA uses a compressed binary format.

```
LOAD<filename>[,R]
```

Loads a program from a disk file into memory. The R option runs the program immediately.

LOAD deletes the current contents of memory and closes all files before LOADing. If R is included, however, open data files are kept open. Programs can be chained or loaded in sections and still access the same data files.

LOAD < filename > ,R and RUN < filename > ,R are equivalent.

```
RUN<filename>[,R]
```

Loads the program from disk into memory and runs it. RUN deletes the current memory contents and closes all files before loading the program. If the R option is included, all open data files are kept open.

RUN < filename > ,R and LOAD < filename > ,R are equivalent.

```
MERGE<filename>
```

Loads the program from disk into memory without deleting the current memory contents. Line numbers used by the disk program are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the merged program resides in memory, and VBASICA returns to command level.

```
KILL<filename>
```

Deletes the specified file from the disk. < filename > can be a program file, or a sequential or random access data file.

```
NAME <old filename> AS <new filename>
```

Changes the name of a disk file. NAME can be used with program files, random files, or sequential files.

G.2 Protected File

To save a program in an encoded binary format, use the P (Protect) option with the SAVE command:

```
SAVE "MYPROG",P
```

A program saved in this way cannot be listed or edited. You may also want to save an unprotected copy of the program for listing and editing purposes.

G.3 Disk Data Files—Sequential and Random I/O

Two types of disk data files are created and accessed by a VBASICA program: sequential files and random access files.

G.3.1 Sequential Files

Although sequential files are easier to create than random files, they are not as fast or flexible when accessing the data. Data written to a sequential file is a series of ASCII characters stored one item after another (sequentially) in the order it is sent. Data is read back the same way.

The statements and functions used with sequential files are:

OPEN
CLOSE
PRINT#, PRINT# USING
EOF
INPUT#, LINE INPUT#
LOC
WRITE#

Follow these steps to recreate a sequential file and access the data in the file:

1. OPEN the file in O mode:

```
OPEN "O",#1,"DATA"
```

2. Write data to the file using the PRINT# statement. (WRITE# can be used instead.)

```
PRINT#1,A$;B$;C$
```

3. To access the data in the file, you must CLOSE the file and reopen it in "I" mode.

```
CLOSE #1
OPEN "I",#1,"DATA"
```

4. Use the INPUT# statement to read data from the sequential file into the program.

```
INPUT#1,X$,Y$,Z$
```

Here is a short program that creates a sequential file, "DATA", from information you enter at the keyboard:

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$="DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
RUN
NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/83

NAME? MANFRED MANN
DEPARTMENT? KEYBOARD REPAIR
DATE HIRED? 08/16/83
    .
    .
    .
```

The next program accesses the file created in the previous example and displays the name of everyone hired in 1983:

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEZER SCROOGE
MANFRED MANN
Input past end in 20
Ok
```

The preceding program reads (sequentially) every item in the file. When all the data is read, line 20 causes an "Input past end" error. To avoid getting this error, insert the following line 15, which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

Then change line 40 to:

```
GOTO 15
```

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement:

```
PRINT#1, USING"####.##,";A,B,C,D
```

can be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

When used with a sequential file, the LOC function returns the number of sectors that have been written to or read from the file since it was OPENed. (A sector is a 128-byte block of data.)

Adding Data to a Sequential File

If you want to add data to the end of a sequential file residing on disk, you cannot simply open the file in O mode and start writing data. If you do this, you destroy the current contents of the sequential file. Instead, use the following procedure (used here to add data to an existing file called NAMES):

1. OPEN NAMES in Insert mode.
2. OPEN a second file called COPY in O mode.
3. Read in the data in NAMES and write it to COPY.
4. CLOSE NAMES and KILL it.
5. Write the new information to COPY.
6. Rename COPY as NAMES and CLOSE.
7. Now there is a disk file called NAMES that includes all previous data plus the new data you just added.

The next example illustrates this technique:

```
10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO "COPY"
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE #1
100 KILL "NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$=""THEN 200 'CARRIAGE RETURN EXITS
     INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#2,N$
170 PRINT#2,A$
180 PRINT#2,B$
190 PRINT:GOTO 120
```

```
200 CLOSE
205 REM CHANGE FILENAME BACK TO "NAMES"
210 NAME "COPY" AS "NAMES"
2000 IF ERR=53 AND ERL=20 THEN OPEN
      "O",#2,"COPY":RESUME 120
2010 ON ERROR GOTO 0
```

The preceding program can be used to create or add to a file called NAMES. This program also shows how to use LINE INPUT# to read strings that contain commas from the disk file. Remember, LINE INPUT# reads in characters from the disk until it sees a carriage return or it has read 255 characters. (It does not stop at quotation marks or commas.)

The error-trapping routine in line 2000 traps a "File does not exist" error in line 20. When this happens, the statements that copy the file are skipped, and COPY is created as a new file.

G.3.2 Random Files

You need more program steps to create and access random files than you do with sequential files; however, there are advantages to taking the extra trouble. One advantage is that random files require less room on the disk; VBASICA stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.) The biggest advantage to random files is that data is accessed randomly. You don't have to read through all the information, as you do with sequential files. Random access is possible because the information is stored and accessed in distinct units called records. Each record is numbered, making it easy for VBASICA to locate the record you need.

The statements and functions used with random files are:

OPEN
PUT
MKI$, MKS$, MKD$
FIELD
CLOSE
CVI, CVS, CVD
LSET/RSET
LOC
GET

Creating a Random File

Use these program steps to create a random file.

1. OPEN the file for random access (R mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.

```
OPEN "R",#1,"FILE",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables to be written to the random file.

```
FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
```

3. Use LSET to move the data into the random buffer. Numeric values must be put into strings when put in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string; MKS$ for a single-precision value; and MKD$ for a double-precision value.

```
LSET N$=X$
LSET A$=MKS$(AMT)
LSET P$=TEL$
```

4. Finally, use the PUT statement to write data from the buffer to the disk:

```
PUT #1,CODE%
```

The next example accepts input data from the keyboard and writes it to a random file:

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

Each time the PUT statement is executed, a record is written to the file. The two-digit code input in line 30 becomes the record number.

Do not use a FIELDed string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

Accessing a Random File

These program steps are required to access a random file:

1. OPEN the file in "R" mode:

```
OPEN "R",#1,"FILE",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.

```
FIELD #1 20 AS N$, 4 AS A$, 8 AS P$
```

   If a program does input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer:

```
GET #1,CODE%
```

4. The data in the buffer can now be accessed by the program. Convert numeric values back to numbers using the "convert" functions: CVI for integers; CVS for single-precision values; and CVD for double-precision values:

```
PRINT N$
PRINT CVS(A$)
```

Using this procedure, you can use a three-digit code to access and display records in the random file "FILE" created in the last example:

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1,CODE%
50 PRINT N%
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

With random files, the LOC function returns the current record number. The current record number is computed by adding one to the number of the last record used in a GET or PUT statement. For example, this statement ends program execution if the current record number in file 1 is higher than 50:

```
IF LOC(1)>50 THEN END
```

The next program is an inventory program that uses random file access. In this program, the record number is also the part number, and it is assumed the inventory uses no more than 100 different part numbers. Lines 900–960 initialize the data file by writing CHR$(255) as the first character of each record. This character is used later (line 270 and line 500) to determine whether an entry already exists for that part number. Lines 130–220 show the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine. Here is the inventory program:

```
120 OPEN "R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$,2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION<1)OR(FUNCTION>6) THEN PRINT
    "BAD FUNCTION NUMBER":GO TO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)>255 THEN INPUT"OVERWRITE";A$:
    IF A$>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
```

```
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC (F$)-255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE$$##.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT"QUANTITY TO ADD";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT #1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%;"IN STOCK":GOTO 600
630 Q%=Q%-S%
640 IF Q%=<CVI(R$) THEN PRINT"QUANTITY NOW";Q%;
    "REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)CVI(R$) THEN PRINT D$;"QUANTITY";
    CVI (Q$)TAB(50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100)THEN PRINT "BAD PART
    NUMBER":GOTO 840 ELSE GET #1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IFB$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

Index